BiqCrunch online Solver User's Guide

Marco Casazza and Frédéric Roupin *

July 27, 2012

*LIPN - CNRS UMR7030 - Université Paris-Nord, Sorbonne Paris Cité
Frederic.Roupin@lipn.univ-paris13.fr

Summary

BiqCrunch is a semidefinite-based solver for binary quadratic problems. It uses a branch-and-bound method featuring an improved semidefinite bounding procedure [8], mixed with a polyhedral approach (see [4, 5] for details). *BiqCrunch* is written in `C` and `Fortran` and uses the library *L-BFGS-B* for quasi-Newton bound-constrained optimization [2, 13], and the Branch-and-Bound framework *BOB*[7].

People involved with the development of BiqCrunch are

- Nathan Krislock (`nathan.krislock@inria.fr`)
- Jérôme Malick (`jerome.malick@inria.fr`)
- Frédéric Roupin (`Frederic.Roupin@lipn.univ-paris13.fr`)

1 BiqCrunch

For now *BiqCrunch* is available only as online solver.
You can try it at http://www-lipn.univ-paris13.fr/BiqCrunch/solver.

1.1 Usage

The *BiqCrunch* online solver is very simple to use but you have to provide some information to be able to solve your problem:

e-mail address: a valid e-mail address where *BiqCrunch* online solver will send the results of the computation;

objective: you can either choose to maximize or minimize the value of the objective function;

specific problem heuristic: you can choose the heuristics used during the computation. The online solver offers you a generic heuristic and some heuristics for specific problems;

verbosity level: this flag allows the user to choose the level of details given with the output. Right now you can choose between a low detailed output or a more completed output with informations about the bound computation at each node of the search tree;

upload method: the instance of the problem can be provided uploading a file or coding the instance directly in the website.

Note that there are some limitations: the online solver handles problems with no more than 100 variables and 5000 constraints. The time limit of computation is one hour, after that the user will receive the best solution found so far and the current gap. The input file uploaded on the website must be in `BC` format and weigh less than 1MB; Finally, when submitting the problem, the user agrees that the data will be stored and may be added with a credit to the *BiqCrunch* library of users instances.

1.1.1 BiqCrunch Parameters

The user can also modify the behaviour of *BiqCrunch* by changing the values of some parameters:

root: by default *BiqCrunch* solves the problem to optimality but checking this option you can ask to *BiqCrunch* to stop the branch-and-bound after the evaluation of the root node;

withCuts: by default *BiqCrunch* adds and removes dynamically cuts (see [4]) to improve the bound during the computation. This option generally offers much better performance but you can also try *BiqCrunch* without this feature;

heur_1: this parameter asks to *BiqCrunch* to use a heuristic before starting the Branch-and-Bound to get a feasible solution. The heuristic used depends on the problem chosen by the user[1];

heur_2: this parameter asks to *BiqCrunch* to use a heuristic to try to improve the best current feasible solution during the computation of the bound of each node of the branch-and-bound tree. The heuristic used depends on the problem chosen by the user[1];

heur_3: this parameter asks to *BiqCrunch* to use a heuristic after the evaluation of a node of the branch-and-bound tree. The heuristic used depends on the problem chosen by the user[1];

gapCuts: minimum value to consider a triangle inequality as violated. The value must be between -1 and 0. By default *BiqCrunch* uses a gap of -5e-2;

1.2 Input format

BiqCrunch allows to solve problems that can be stated as

$$\begin{array}{rll} \max & x^T A_0 x + b_0^T x + c_0 & \\ \text{subject to} & x^T A_i x + b_i^T x = c_i & \forall i \in \{1, \ldots, m_E\} \\ & x^T A_j x + b_j^T x \leq c_j & \forall j \in \{1, \ldots, m_I\} \\ & x \in \{0,1\}^n & \end{array}$$

The problem has to be converted in `BC` format, very similar to standard `SDPA` format (see Instance 1.1). In the `BC` format, the problem is described in matrix form by considering the matrix $X = \begin{bmatrix} xx^T & x \\ x^T & 1 \end{bmatrix}$, and defining the inner product of two matrices X and Y by $X \bullet Y = Trace(X^T Y)$. In particular, the objective function is written as $Q_0 \bullet X$ where $Q_0 = \begin{bmatrix} A_0 & \frac{b_0}{2} \\ \frac{b_0^T}{2} & c_0 \end{bmatrix}$. Similarly, each constraint $k \in \{1, \ldots, m_I + m_E\}$ is associated to a matrix Q_k such that $Q_k = \begin{bmatrix} A_k & \frac{b_k}{2} \\ \frac{b_k^T}{2} & 0 \end{bmatrix}$. As done in `SDPA` format, the right-hand side c of the constraints is stored aside. In the next section, we precise how we indicate that a constraint is an equality or an inequality.

[1] see section 2 for further explanation

```
<COMMENT>
      ⋮
<COMMENT>
<#CONSTRAINTS> =number of constraints
<#BLOCKS> =blocks of the matrices
<SIZE>
<RIGHT-HAND_SIDE>
<#MATRIX> <BLOCK> <ROW_INDEX> <COLUMN_INDEX> <VALUE>
      ⋮
<#MATRIX> <BLOCK> <ROW_INDEX> <COLUMN_INDEX> <VALUE>
```

Code 1.1: Generic structure of a BC instance

1.2.1 Instance syntax

A BC file begins with some optional lines of comments, which are strings preceded by a semicolon or by a asterisk:

```
<COMMENT> ::= ; <STRING> | * <STRING>
```

The first line after the comments defines the number of constraints in our model, which is a integer non-negative number that can be followed by other characters ignored by the reader

```
<#CONSTRAINTS> ::= <INT> | <INT> <STRING>
```

and such that <INT>$= m_E + m_I$.

As we keep the syntax of the SDPA files, we define the number of blocks of the matrices of the input file. As seen before, also this line admit characters after the definition.

```
<#BLOCKS> ::= <INT> | <INT> <STRING>
```

In the BC an instance can have 1 or 2 blocks depending on the model: if the model contains no constraints or just equality constraints, <INT> must be equal to 1; if the model contains also inequality constraints <INT> must be equal to 2.

The third entry of the instance describes the size of the blocks of the matrices

```
<SIZE> ::= <INT_1>           | {<INT_1>}           |
           <INT_1>, -<INT_2> | {<INT_1>, -<INT_2>}
```

If we describe a problem without inequalities, we have only to provide the size of the first block of the matrices (<INT_1>), which is equal to $n + 1$. If the problem contains inequalities we have to define also the size of the second block (<INT_2>), which starts always with a *minus* before its size, to explicit that the values of the blocks are only on the diagonal of the matrix (since they correspond to *slack* variables), and that must be equal to m_I.

In the next line we have to define the right-hand side values of the constraints which is a sequence of values

```
<RIGHT-HAND_SIDE> ::= <REAL_k> | <REAL_k> <RIGHT-HAND_SIDE>
```

The number of values must be equal to $m_E + m_I$ and `<REAL_k>` must be the right-hand side value of constraint k.

At the end of the instance we define all the matrices that describe the objective function and the left-hand side of the constraints. The first matrix to describe is the objective function coefficient matrix Q_0: we describe the matrix in sparse format, and we have to write a line for every non-zero element of the matrix

```
<OBJ_MATRIX_EL> ::= 0 1 <INT_1> <INT_2> <REAL>
```

where the first and second number means that we're defining the objective function matrix and we're defining the first block. `<INT_1>` and `<INT_2>` are the row and the column of the element of the matrix and `<REAL>` is his value. `<INT_1>` and `<INT_2>` must be greater than 0 and less or equal to $n + 1$.

After the Q_0 matrix we have to describe all the constraints matrices, and for each constraint k we have to write the values of the matrix Q_k in sparse format:

```
<CONS_MATRIX_EL> ::= <INDEX_k> 1 <INT_1> <INT_2> <REAL>
```

where `<INDEX_k>` must be equal to k.

If the constraint we're defining is an inequality j, we have to define the value of the second block of the matrix:

```
<INEQ_MATRIX_EL> ::= <INDEX_k> 2 <INT> <INT> <REAL>
```

where `<INT>` must be equal to j and real can be 1.0 if the inequality is in form of $\leq$ or -1.0 if it's a $\geq$ inequality.

1.2.2 Examples

We report now a couple of examples to understand how to formulate `BC` instances. You can copy these examples and solve them using the *from source* option in the online solver at http://www-lipn.univ-paris13.fr/BiqCrunch/solver.

Model 1

$$\begin{aligned} \max \quad & x_1x_3 + x_1x_4 + x_2x_3 + x_2x_5 + x_3x_4 + x_4x_5 \\ \text{subject to} \quad & x_1 + x_2 + x_3 + x_4 + x_5 = 3 \\ & x \in \{0,1\}^n \end{aligned}$$

```
; Example of an instance for BiqCrunch
; This instance contains only an equality
1 = #constraint
1 = just equalities
6
3.0
0 1 1 3 0.5
0 1 1 4 0.5
```

```
0 1 2 3 0.5
0 1 2 5 0.5
0 1 3 4 0.5
0 1 4 5 0.5
1 1 1 6 0.5
1 1 2 6 0.5
1 1 3 6 0.5
1 1 4 6 0.5
1 1 5 6 0.5
```

Model 2

$$\begin{aligned} \max \quad & 20x_1x_3 + 26x_1x_4 + 23x_2x_3 + 8x_2x_5 + 32x_3x_4 + 13x_4x_5 \\ \text{subject to} \quad & x_1 + x_2 + x_3 + x_4 + x_5 = 3 \\ & 12x_1x_3 + 24x_1x_4 + 14x_2x_3 + 16x_2x_5 + 28x_3x_4 + 12x_4x_5 \leq 30 \\ & x \in \{0,1\}^n \end{aligned}$$

```
; Example of an instance for BiqCrunch
; This instance contains equalities and inequalities
2 =constraint
2 =equalities and inequalities
6, -1
3.0 30.0
0 1 1 3 10.0
0 1 1 4 13.0
0 1 2 3 11.5
0 1 2 5 4.0
0 1 3 4 16.0
0 1 4 5 6.5
1 1 1 6 0.5
1 1 2 6 0.5
1 1 3 6 0.5
1 1 4 6 0.5
1 1 5 6 0.5
2 1 1 3 6.0
2 1 1 4 12.0
2 1 2 3 7.0
2 1 2 5 8.0
2 1 3 4 14.0
2 1 4 5 6.0
2 2 1 1 1.0
```

2 Heuristics

The *BiqCrunch* online solver uses some heuristics to improve the performance of the branch-and-bound by building feasible solutions (generally from the SDP solution provided by the bounding procedure). We have three types of heuristics:

root-node heuristic: called once before starting the Branch-and-Bound;

bound heuristic: called a large number of times while computing the bound of the branch-and-bound nodes;

node heuristic: called after the evaluation of eachnode of the branch-and-bound tree.

We provide different versions of *BiqCrunch* which use also heuristics for specific problems.

2.1 Generic problem

For generic problems *BiqCrunch* provides some general heuristics that can improve the performance of the branch-and-bound by trying to build a feasible 0-1 vector for the combinatorial problem from the SDP solution. These heuristics can be used for any quadratic 0-1 problem.

During the computation of the bound of the node and after the evaluation of each node, *BiqCrunch* uses a variant of the classical randomized rounding heuristic [10, 9] that rounds to 1 the variables according to the probability provided by the fractional SDP solution. Indeed one has $0 \leq x_i \leq 1$ for any feasible solution of the SDP relaxation (see [8] for details about the relaxations used). This is done simply by comparing these values to a fixed one α which goes from 0 to 1 by a step of 0.01. Then we test if the resulting 0-1 vector is feasible for the combinatorial problem, and update the best current feasible solution if needed.

At root node we generate a random vector of values in $[0, 1]$ domain and then we apply the variant of randomized algorithm described before.

2.2 Max-Cut problem

Given an edge-weighted graph G with n vertices and edge weights w_{ij} for $(ij) \in E$, the objective is to maximize the total weight of the edges between a subset of vertices and its complement (see [4] for details). It can be stated as:

$$\begin{aligned} \max \quad & \sum_{i,j} w_{ij} x_i (1 - x_j) = Q \bullet x x^T \\ \text{subject to} \quad & x \in \{0,1\}^n \end{aligned}$$

Note that for max-cut ising problems (see e.g.[4]), we provide a specific option in the online solver (using a particular parameter setting).

2.2.1 Max-Cut heuristic

BiqCrunch for max-cut problems uses two different heuristics:

- the same randomized rounding heuristic as the generic *BiqCrunch* during the computation of the bound at each node of the search tree;
- the Goemans-Williamson random hyperplane algorithm [3] after the evaluation of each node.

2.2.2 BiqCrunch instances and conversion

An instance for the max-cut problem should be in a format like the example in Instance 2.1, defining only the objective function coefficient matrix.

```
0 =number of constraints
1 =blocks of the matrices
n
0 1 i j Q_ij
    ⋮
```

Code 2.1: Structure of a `BC` instance for *Max-Cut* problems

To obtain *BiqCrunch* input files for the max-cut or Unconstrained 0-1 quadratic problems (see e.g. [1]) you can simply use the conversion tools available at `http://www-lipn.univ-paris13.fr/BiqCrunch/Download` which can create instances from the rudy format [11] to `BC`.

2.3 K-cluster problem

Given a graph $G = (V, E)$ the k-cluster problem consists of determining a subset $S \subseteq V$ of k vertices such that the sum of the weights of the edges between vertices in S is maximized.

Letting $n = |V|$ denote the number of vertices, and w_{ij} denote the edge weight for $(ij) \in E$ and $w_{ij} = 0$ for $(ij) \notin E$, the problem can be modelled as the following 0-1 quadratic problem:

$$\begin{aligned} \max \quad & \frac{1}{2} x^T W x \\ \text{subject to} \quad & \sum_i x_i = k \qquad\qquad x \in \{0,1\}^n \end{aligned}$$

where $W = (w_{ij})_{ij} \in \mathbb{S}^n$ is the weighted adjacency matrix of the graph G.

2.3.1 K-cluster heuristic

We use two types of heuristics to find a cluster with exactly k nodes. First, for the initial feasible point (before running the Branch-and-Bound), we use the classical greedy heuristic, since it gives very good feasible solutions: we remove one by one vertices from the graph by choosing at each step the one with the smallest degree (or sum of the weights over the adjacent vertices). Second, during the evaluation of the bound and after running the bounding procedure on a subproblem having k' nodes added to the cluster, we add the remaining $k - k'$ nodes having the largest fractional values x_i in the SDP solution. More detailed can be found in [5].

2.3.2 K-cluster instances and conversion

An instance for the k-cluster problem should be in a format like the example in Instance 2.2, defining the objective function coefficient matrix and the equality constraint.

```
1 =number of constraints
1 =blocks of the matrices
n+1
k
0 1 i j W_ij
    ⋮
0 1 i j W_ij
1 1 1 n+1 0.5
    ⋮
1 1 n n+1 0.5
```

Code 2.2: Structure of a `BC` instance for k-cluster problems

To obtain *BiqCrunch* input files for the k-cluster problem you can simply use the conversion tools available at `http://www-lipn.univ-paris13.fr/BiqCrunch/Download` which can convert instances from Rudy format and from the format used in [6] to `BC`. When using the conversion tool, weights can be ignored with a simple flag if the graph is unweighted. Note that the conversion tools add also redundant constraints to the instance to improve the bound obtained during the bounding procedure [12]. A user guide to the conversion tool is included in the archive available online.

Bibliography

[1] Alain Billionnet and Sourour Elloumi. Using a mixed integer quadratic programming solver for the unconstrained quadratic 0-1 problem. *Mathematical Programming*, 109:55–68, 2007. 10.1007/s10107-005-0637-9.

[2] Richard H. Byrd, Peihuang Lu, Jorge Nocedal, and Ciyou Zhu. A limited memory algorithm for bound constrained optimization. *SIAM J. Sci. Comput.*, 16(5):1190–1208, September 1995.

[3] Michel X. Goemans and David P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *J. ACM*, 42(6):1115–1145, November 1995.

[4] Nathan Krislock, Jerome Malick, and Frédéric Roupin. Improved semidefinite bounding procedure for solving Max-Cut problems to optimality. Available at http://hal.archives-ouvertes.fr/hal-00665968. Submitted to Mathematical Programming.

[5] Nathan Krislock, Jérôme Malick, and Frédéric Roupin. Improved semidefinite branch-and-bound algorithm for k-cluster. Available at http://hal.archives-ouvertes.fr/hal-00717212. Submitted to Journal Of Combinatorial Optimization.

[6] Amélie Lambert. A library of k-cluster problems. CNAM-CEDRIC, http://cedric.cnam.fr/ lamberta/Library/k-cluster.html.

[7] Bertrand Le Cun, Catherine Roucairol, and The Pnn Team. Bob: a unified platform for implementing branch-and-bound like algorithms. Technical report, Laboratoire Prism, 1995.

[8] Jérôme Malick and Frédéric Roupin. On the bridge between combinatorial optimization and nonlinear optimization: a family of semidefinite bounds for 0-1 quadratic problems leading to quasi-Newton methods. Technical report, 2011. Available at http://hal.archives-ouvertes.fr/hal-00662367. To appear in Mathematical Programming.

[9] Prabhakar Raghavan. Probabilistic construction of deterministic algorithms: approximating packing integer programs. *J. Comput. Syst. Sci.*, 37(2):130–143, October 1988.

[10] Prabhakar Raghavan and Clark D. Tompson. Randomized rounding: a technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7(4):365–374, December 1987.

[11] Giovanni Rinaldi. Rudy, a graph generator. http://www-user.tu-chemnitz.de/ helmberg/rudy.tar.gz.

[12] Frédéric Roupin. From linear to semidefinite programming: An algorithm to obtain semidefinite relaxations for bivalent quadratic problems. *Journal of Combinatorial Optimization*, 8:469–493, 2004. 10.1007/s10878-004-4838-6.

[13] Ciyou Zhu, Richard H. Byrd, Peihuang Lu, and Jorge Nocedal. Algorithm 778: L-bfgs-b: Fortran subroutines for large-scale bound-constrained optimization. *ACM Trans. Math. Softw.*, 23(4):550–560, December 1997.